Informatik II – Datenstrukturen und Algorithmen in C/C++ — Nachtrag

Dr. Stefan Lewandowski[1]

Version 25. Juni 2011

Dieses Skript darf nur mit ausdrücklicher Zustimmung des Autors vervielfältigt werden.
Die Weiterverwendung des Skripts (auch in Auszügen) und der zugehörigen Programme
bedarf der Zustimmung des Autors.

B.5½ Speichern und Laden von binären Suchbäumen mit Hilfe der Preorder

Wir haben in der Vorlesung gelernt:

- Fügt man die Elemente eines binären Suchbaums in der Preorder-Reihenfolge in einen anfangs leeren Baum ein, so erhält man wieder denselben Baum.
- Der Aufwand für das Einfügen benötigt pro Element $O(n)$, das gesamte Einfügen der Elemente mit dem Standard-Einfügeverfahren kann also quadratisch viel Zeit benötigen (überlegen Sie sich mind. 3 Reihenfolgen für solche binären Suchbäume mit den Elementen 1 bis 7). Für die Praxis ist dies zu langsam!
- Im Mittel bleibt die Höhe eines (zufälligen) binären Suchbaums logarithmisch beschränkt. Die erwartete Laufzeit ist somit proportional zu $n \log n$.

Ziel dieser Aufgabe ist, einen Linearzeitalgorithmus für das Erstellen eines binären Suchbaums aus dessen Preorder-Reihenfolge zu entwickeln – zur Vorbereitung sind hier einige Vorüberlegungen notwendig, um diese Laufzeit für beliebige binäre Suchbäume garantieren zu können. Nebenbei wollen wir in dieser Aufgabe kennenlernen, wie wir mit C++ in Dateien schreiben und von Dateien lesen können. Wer sich um die Dateioperationen später kümmern möchte kann den nächsten Abschnitt erstmal überspringen.

Schreiben und Lesen von Textdateien: In C++ werden Dateien standardmäßig als Textdateien interpretiert, auf die man mit den von cout und cin bekannten Operatoren << und >> zugreifen kann. Zuvor muss man die Datei zum Lesen und / oder Schreiben öffnen. Zu diesem Zweck werden Variablen vom Typ[2] fstream verwendet. Das Schreiben und Lesen verhält sich identisch wie cout und cin auf der Konsole – d. h. auch, dass beim Lesen Leerzeichen, Tabulator und Zeilenvorschub als Trennsymbol interpretiert werden.

Lesen und Schreiben (Textdatei) – erster Versuch

```
#include<iostream>
#include<fstream> // für Dateizugriffe

using namespace std;

int main(void)
{
  fstream meinDateizugriff;

  cout << "testdatei.txt␣erzeugen" << endl;
```

[1] E-Mail: slewand@slewand.de

[2] Im Sinne von C++ müsste es statt Variable und Typ richtig Objekt bzw. Klasse heißen.

```
  meinDateizugriff.open("testdatei.txt",ios::out);
  // .open(..) öffnet die Datei — Vorsicht, eine evtl. vorhandene Datei
  // wird ohne Rückfrage gelöscht/überschrieben
  meinDateizugriff << 31 << 41 << 59 << 26 << endl;
  // 4 Zahlen schreiben wie bei cout << ..
  meinDateizugriff.close(); // wenn fertig, mit .close() wieder schließen

  cout << "folgende Zahlen wurden geschrieben" << endl;
  int zahl;
  meinDateizugriff.open("testdatei.txt",ios::in);
  // Wiederverwendung der "Variable" meinDateizugriff
  while (!meinDateizugriff.eof()) {
    // solange Dateiende nicht (!) erreicht ist (. End Of File ())
    meinDateizugriff >> zahl; // wie cin >> zahl, bloß hier aus Datei lesen
    cout << zahl << endl; // zahl "verarbeiten"
  }
  meinDateizugriff.close();

  return 0;
}
```

Testet man dieses Programm, sieht man sich mit zwei Problemen konfrontiert:

- die Zahlen werden nicht einzeln wieder eingelesen
- die „letzte“ Zahl wird doppelt eingelesen

Das Verhalten ist im Wesentlichen identisch dem bei der Eingabe von der Konsole. Die Zahlen waren nicht voneinander getrennt – dementsprechend wurde die Zeichenfolge `31415926` als eine Zahl interpretiert (zum Vergleich die testdatei.txt mit einem Editor öffnen). Der zweite Punkt ist schwerer nachzuvollziehen: versuchen Sie hierzu einmal mit cin >> zahl eine Zahl von der Konsole einzulesen und tippen dabei `42x` ein. Die Variable zahl wird danach den Wert `42` haben. Folgt im Programm ein cin >> buchstabe (für eine Variable **char** buchstabe), so wird buchstabe ohne weitere Eingabe mit dem Wert `x` gefüllt. Würde man an der Stelle stattdessen ein weiteres **int** einlesen, so bekäme dieses ebenfalls den Wert `42` (Grund siehe weiter unten).

Was ist passiert? Das Einlesen erfolgt immer bis zum ersten Zeichen, das nicht zum Datentyp passt, der Rest bleibt im Puffer zum Einlesen hängen. Passt bereits das erste Zeichen nicht zum Datentyp, so ist das Einlesen nicht erfolgreich (dies könnte man bei cin mit einer Abfrage **if** (cin.fail()) { ... } abfangen, der letzte Fehler bleibt bis zu einem Aufruf von cin.clear() erhalten, bis dahin bleiben ohne weitere Maßnahmen alle cin erfolglos) – analog funktioniert dies auch bei fstream: in unserem Programm hätte man nach dem Einlesen mit **if** (meinDateizugriff.fail()) { ... } entsprechend handeln können. Das Einleseobjekt (cin beim Einlesen von der Konsole oder meinDateizugriff bei unserem Beispiel des Dateizugriffs) gibt den Fehler auch direkt als boolschen Rückgabewert zurück. Man kann so z. B. ein fehlerhaftes Einlesen direkt mit **if** (cin >> zahl) { */*erfolgreich */*} **else** { */*nicht erfolgreich */*} verarbeiten (im Fehlerfall cin.clear() nicht vergessen).

Und warum wird nun der Wert `42` zweimal eingelesen? Dies wird er nicht, das letzte Einlesen schlägt fehl (das erste Zeichen passt nicht zum Datentyp **int** – bei dem Beispiel der Konsole ist dies das `x`, bei der Datei der Zeilenvorschub, der am Dateiende übrig blieb), dabei bleibt der Wert der Variablen zahl unverändert – probieren Sie auf der Konsole z. B. mit Eingabe `42x` zwei **int** und ein **char** einzulesen, sie erhalten dabei zweimal den Wert `42` und danach

den Buchstaben, den die Variable vor(!) dem Einlesen hatte – das zweite (fehlerhafte) cin hätte beim Einlesen mit dem Rückgabewert **false** oder mittels cin.fail() abgefangen werden können, weitere cin bleiben erfolglos, bis cin.clear() aufgerufen wird (danach kann das immer noch im Puffer vorhandene `x` eingelesen werden). Warum bei einer Datei mit jeweils durch Zeilenvorschub getrennten Zahlen dieses Problem nur bei der letzten Zahl auftritt, liegt daran, dass hierbei nach dem Zeilenvorschub jeweils noch eine Zahl kommt (wie bei der Konsole, die nach einem Zeilenvorschub die dann kommende Eingabe noch akzeptiert) – bis auf den Fall der letzten Zahl, nach der nichts mehr kommt.

Eine simple Lösung für den vorliegenden Fall wäre, einfach solange Zahlen einzulesen, wie dies erfolgreich ist, also

Lesen (Textdatei) – zweiter Versuch

```
while (meinDateizugriff >> zahl) { // solange Lesen erfolgreich ist
  cout << zahl << endl; // zahl "verarbeiten"
}
```

Will man nur lesend oder nur schreibend auf eine Datei zugreifen, so kann man ohne Angabe von ios::in oder ios::out statt fstream den Typ ifstream bzw. ofstream verwenden. Im obigen Beispiel hätte man dann aber die Variable meinDateizugriff nicht wiederverwenden können, da diese zuvor als ofstream deklariert worden wäre, danach also nicht plötzlich ifstream sein kann (vergleichbar mit dem Versuch ein **int** zahl; danach als **float** zahl; neu zu deklarieren).

Binärdateien: Beim Abspeichern von Datenstrukturen wird man diese meist nicht als Einzelzeichen abspeichern wollen sondern so platzsparend, wie sie auch im Speicher des Rechners vorliegen. Dazu muss man die Dateien beim Öffnen mittels des zusätzlichen Parameters ios::binary als Binärdatei kenntlich machen (dieser zusätzliche Parameter wird ggf. mittels der Verknüpfung | mit z. B. ios::out verbunden). Zum Lesen und Schreiben werden statt >> und << nun die Funktionen read bzw. write verwendet. Diese erwarten als Parameter jeweils eine Adresse im Speicher (als Pointer auf ein **char**-Array) und die Anzahl der zu lesenden / schreibenden Bytes. Da wir in der Regel nicht Daten in ein **char**-Array einlesen wollen, müssen wir den Pointer explizit umwandeln, um Compiler-Fehler oder -Warnungen zu vermeiden (siehe Programmcode). Ebenso kann die Größe der Standarddatentypen vom Compiler abhängen, man sollte daher diese Anzahl mittels **sizeof** ermitteln.[3]

Bei den zu lesenden Daten sind wir nicht darauf beschränkt, einzelne Variablen einzulesen. Wir können alles, was im Speicher später direkt hintereinander steht, auch „in einem Rutsch“ einlesen, im folgenden Beispiel die gesamte Datei, die in ein Array eingelesen wird[4].

Lesen und Schreiben (Binärdatei)

```
#include<iostream>
#include<fstream> // für Dateizugriffe

using namespace std;
5
```

[3] Dies wirkt unschön, ist es letzten Endes auch, die Alternative von C mittels printf und scanf sieht jedoch nur auf den ersten Blick schöner aus. In C++ hat man prinzipiell noch die Möglichkeit die Operatoren << und >> zu überschreiben und so gleichermaßen für Text- und Binärdateien tauglich machen (dies lohnt die Mühe jedoch nicht – wir bleiben bei read und write).

[4] Bzgl. der Laufzeit ist dies sehr vorteilhaft, da ein ständiges Neupositionieren des Lesekopfes der Festplatte entfällt – dieses kann sonst zu einer vielfachen Zeit zum Einlesen der Daten vom Datenträger führen.

```
void generatePreorder(fstream & file,
                      const int intervalLeft, const int intervalRight)
{ // Preorder eines binären Suchbaums in eine mit ios::binary geöffnete
  // Datei schreiben
  // um wiederholbare Ergebnisse zu erhalten, den Zufallszahlengenerator
  // ggf. mittels srand() vor dem Aufruf zur
  // Preorder-Generierung mit einem festen Wert initialisieren
  if(intervalRight>=intervalLeft)
  {
    // zufällige Wurzel wählen, danach erst den linken,
    // dann den rechten Unterbaum schreiben
    int rootvalue = rand()%(intervalRight-intervalLeft+1) + intervalLeft;
    file.write((char*)&rootvalue,sizeof(rootvalue)); // charPointer+Size
    generatePreorder(file,intervalLeft,rootvalue-1);
    generatePreorder(file,rootvalue+1,intervalRight);
  }
}

int main(void)
{
    cout << "testdatei.bin erzeugen" << endl;

    fstream file("testdatei.bin",ios::binary|ios::out);
    // statt eines separaten .open(...) kann man bei der Deklaration
    // auch gleich den Dateinamen und Öffnungsmodus angeben
    generatePreorder(file,1,25);
    file.close();

    cout << "testdatei.bin en block einlesen" << endl;

    file.open("testdatei.bin",ios::binary|ios::in);
    file.seekg(0,ios::end); // zur Position Byte 0 vom Ende der Datei aus
    int abssize=file.tellg(); // Position auslesen = Größe der Datei
    int intsize=abssize/sizeof(int); // Dateigröße in Anzahl der ints
    file.seekg(0,ios::beg); // wieder an Pos. 0 springen
    int *a = new int[intsize];
    file.read((char*)a,abssize); // alles auf einmal einlesen
    file.close();

    int i;
    for(i=0;i<intsize;i++)
    {
      cout << a[i] << endl;
    }

    delete[] a;

    return 0;
}
```

Jetzt können wir eine Reihe von Zahlen (wie hier eine zufällig erzeugte Preorder-Reihenfolge eines binären Suchbaums) abspeichern und wieder einlesen und uns wieder der eigentlichen Aufgabe widmen.

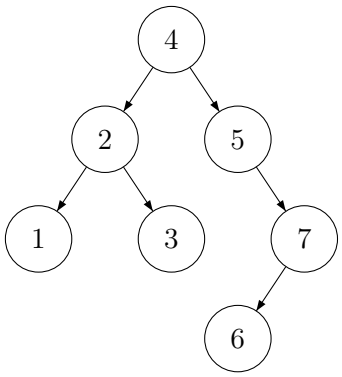
Randbemerkung 1: Wenn Sie versuchen Binärdateien mit einem Editor zu öffnen, werden Sie

feststellen, dass Sie diese nicht ohne Weiteres lesen können. Wenn ein Lesen und Verändern von Hand nötig ist, muss man auf Textdateien ausweichen (mit den Problemen, die dies dann von der Handhabung mit sich bringt).

Randbemerkung 2: Dies alles zeigt nur einen kleinen Ausschnitt aus den Befehlen, die bei der Dateiverarbeitung eine Rolle spielen (können). Bei Interesse bitte in einem Buch oder im Netz selbstständig weiterforschen, um z. B. Fehler beim Öffnen von Dateien abzufangen oder das Überschreiben bereits vorhandener Dateien zu vermeiden. Ebenso, falls Sie Dateioperationen lieber mit den C-eigenen Befehlen durchführen möchten.

Erkennen fehlerhafter Preorder-Reihenfolgen: Nicht jede Reihenfolge kann Ergebnis eines Preorder-Durchlaufs sein[5]. Wir wollen hier zunächst überlegen, wie man entscheiden kann, ob eine Reihenfolge eine Preorder-Reihenfolge sein kann. Zur Erinnerung: bei der Preorder-Reihenfolge werden zunächst die Wurzel, danach alle Elemente des linken Unterbaums und dann alle Elemente des rechten Unterbaums ausgegeben. Betrachten wir dazu den folgenden Baum:

Gegeben ist folgender Binärbaum:



Bei der Rekonstruktion des Baums aus dessen Preorder-Reihenfolge „4 2 1 3 5 7 6“ erhalten wir die 4 (die erste Zahl) als Wurzel – alle folgenden Zahlen, die kleiner als 4 sind, gehören in den linken Unterbaum (2, 1 und 3), die, die größer sind, in den rechten (5, 7 und 6). Die Unterbäume werden in gleicher Weise verarbeitet: im linken Unterbaum ist wiederum die erste Zahl, die 2, Wurzel, die 1 dessen linkes und die 3 dessen rechtes Kind; im rechten Unterbaum ist die 5 Wurzel (mit der 7 als rechtem Kind und der 6 als linkem Kind der 7).
Begründen Sie, warum „4 2 3 5 1 7 6“ keine Preorder-Reihenfolge sein kann. An welcher Stelle kann man dies erstmals erkennen?

Überlegen Sie, ob es sinnvoll ist, trotzdem die Elemente in einen binären Suchbaum einfügen zu wollen – betrachten Sie dieselbe Frage ggf. nochmals, nachdem Sie (hoffentlich erfolgreich) den Linearzeit-Algorithmus implementiert haben.

Wir sehen, dass es jeweils eine kleinste Zahl gibt, die noch zu einer gültigen Preorder-Reihenfolge führen kann. Man kann somit die Konsistenz der eingelesenen Daten leicht überprüfen.

Ideenskizze zum Linearzeitalgorithmus: Entscheidend ist, das jeweils nächste Element nicht jeweils von der Wurzel aus einzufügen, sondern die richtige Position von der Position des zuletzt eingefügten Elements aus zu bestimmen. Wir betrachten dazu nochmals das Beispiel von oben: 4 2 1 3 5 7 6 (nehmen Sie ein Blatt und einen Stift zur Hand und malen Sie mit) – die Grundidee wird sein, wenn möglich im linken Unterbaum des zuletzt eingefügten Elements weiterzumachen, anderenfalls wird stets ein rechter Unterbaum begonnen – wir überlegen uns nun, nach welchem Kriterium die richtige Position des Knotens gewählt werden kann.

Der Baum ist zunächst leer, die 4 damit eindeutig die Wurzel, die 2 ist kleiner, muss also in dessen linken Unterbaum, die 1 wiederum kleiner als die 2, kann also von dessen Position aus wieder direkt als linkes Kind eingefügt werden.

[5] Es gibt deutlich mehr verschiedene Reihenfolgen der Zahlen von 1 bis n als verschiedene binäre Suchbäume mit denselben Zahlen. Wieviel mehr? Finden Sie es raus ... Stichwort: *Catalan-Zahlen*.

- Erste Beobachtung: wenn die Zahl kleiner als die vorhergehende ist, muss sie linkes Kind des zuletzt eingefügten Knotens werden.

Insbesondere muss man zum Einfügen nicht nochmals von der Wurzel aus die Position suchen, sondern kann vom zuvor eingefügten Element in konstanter Zeit das aktuelle Element anfügen. Das nächste Element ist nun größer. Wir haben drei Kandidaten, deren rechter Unterbaum bis hierhin noch leer ist.

- Zweite Beobachtung:

  a) zu jedem Knoten ohne rechten Unterbaum gehört ein Intervall möglicher folgender Zahlen

  b) die Intervalle grenzen direkt aneinander – die Werte in den Knoten (zusammen mit ∞ als fiktiven weiteren Wert) bilden die Grenzen der Intervalle

Wir notieren uns die drei Intervalle – wir wissen bereits, dass die Zahl (die 3) größer als die 1 ist. Die Frage ist, bleiben wir noch links von dem nächsten Kandidaten (der 2), oder links von der Wurzel (der 4) oder wird deren rechter Unterbaum begonnen. In diesem Fall liegt die 3 zwischen der 2 und der 4, d. h., der rechte Unterbaum der 2 wird begonnen.

- Dritte Beobachtung: Knoten, die bei dieser Auswahl übersprungen wurden, können auch später keinen rechten Unterbaum mehr bekommen

Dies betrifft hier die 1 – überlegen Sie sich, in welchem Zusammenhang dieses mit der obigen Überlegung zur Erkennung fehlerhafter Preorder-Reihenfolgen steht. Beim Einfügen unserer Zahlen ist nun der zuletzt eingefügte Knoten die 3, neben diesem hat lediglich die 4 noch Chancen einen rechten Unterbaum zu bekommen.

- Vierte Beobachtung: wir können die Knoten, die noch keinen rechten Unterbaum haben, leicht in sortierter Reihenfolge verwalten.

Dies vereinfacht zum einen die Verwaltung der Knoten (ein Stack oder einfach verkettete Liste genügt), zum anderen ist somit die Suche, nach dem richtigen Knoten (zuletzt die 2, die einen rechten Unterbaum bekam) recht einfach.

Die restlichen Schritte laufen analog. Es folgt die 5, diese ist größer als die 3, d. h., wir suchen wieder den richtigen Knoten (samt zugehörigem Intervall), dessen rechter Unterbaum begonnen werden soll. Hier ist die Zahl nun sogar größer als der größte Knoten ohne rechten Unterbaum[6], die 5 wird rechtes Kind der Wurzel und ist nun der einzige Knoten, der noch ein rechtes Kind bekommen kann. Analog wird die 7 rechtes Kind der 5, bevor schließlich die 6 wieder kleiner ist und somit nach obigem Vorgehen linkes Kind des zuletzt eingefügten Elements, also linkes Kind der 7 wird.

[6]Programmiertechnisch kann man diesen Sonderfall vermeiden, indem man in die Liste der Knoten ohne rechten Unterbaum noch einen Pseudoknoten hinzufügt, der den größtmöglichen Wert annimmt. C++ bietet hierfür übrigens sogar ein Programmkonstrukt an: bindet man die Bibliothek #**include**<limits> ein, so steht u. A. der Befehl numeric_limits<**int**>::max() zur Verfügung (ebenso gibt es numeric_limits<**int**>::min() und vieles mehr sowie analoge Befehle für andere Zahltypen – bei Interesse selbst nachforschen).

Aufgaben: Nach diesem langen Prolog wollen wir die Erkenntnisse nun in ein Programm umsetzen, dieses testen, theoretische Betrachtungen zu der Laufzeit anstellen (bisher haben wir nur behauptet, dass das Linearzeit ist) und diese schließlich auch experimentell bestätigen.

- Abstrahieren Sie von obigem Beispiel und schreiben Sie ein Programm, dass aus der Preorder-Reihenfolge eines binären Suchbaums diesen rekonstruiert oder eine fehlerhafte Reihenfolge als solche erkennt.
  - Zum Testen könnte man die Knoten auch mit dem klassischen Verfahren jeweils von der Wurzel aus einfügen und dann die beiden Bäume vergleichen oder einfach die Preorder des konstruierten Baumes bestimmen und diese mit der Ausgangsreihenfolge vergleichen.
- Zur theoretischen Laufzeitabschätzung: Überlegen Sie dazu, wie oft jeder Knoten des Baums beim Einfügen und Vergleichen aus den Intervallgrenzen maximal betrachtet wird.
- Zur Laufzeitmessung und Überprüfung des Linearzeitverhaltens generieren Sie zufällige Bäume mit geeigneter Größe (Sie können dazu obiges Beispielprogramm für das Schreiben und Lesen von Binärdateien verwenden) und messen Sie die Zeit, die Sie zur Konstruktion des Baumes benötigt haben (auf der Vorlesungsseite steht hierzu ein Modul `stopwatch.h/cpp` zur Verfügung[7]). Ein Baum mit 10 mal soviel Elementen sollte also etwa die 10-fache Zeit benötigen. Welchen Faktor würden Sie mit dem Wissen aus der Vorlesung bei dem Standardeinfügeverfahren erwarten?

Versuchen Sie insbesondere auch solche Bäume zu rekonstruieren, die mit den Standardeinfügeverfahren quadratische Laufzeit benötigen (beim Vergleich reichen hier schon etwa 10000 Elemente, um sich von dem Sinn des Linearzeitalgorithmus zu überzeugen).

[7] Es reicht aber auch, bei ausreichend großen Bäumen die Zeit zwischen dem Linearzeitalgorithmus und dem Verfahren mittels klassischem Einfügen zu vergleichen